Sharing and Performance Optimization of Reproducible Workflows in the Cloud[☆]

Rawaa Qasha[a,b], Zhenyu Wen[b], Jacek Cała[b,*], Paul Watson[b]

[a] *College of Computer Sciences and Mathematics, Mosul University, Iraq*
[b] *School of Computing, Newcastle University, United Kingdom*

Abstract

Scientific workflows play a vital role in modern science as they enable scientists to specify, share and reuse computational experiments. To maximise the benefits, workflows need to support the reproducibility of the experimental methods they capture. Reproducibility enables effective sharing as scientists can re-execute experiments developed by others and quickly derive new or improved results. However, achieving reproducibility in practice is problematic – previous analyses highlight issues due to uncontrolled changes in the input data, configuration parameters, workflow description and the software used to implement the workflow tasks. The resulting problems have become known as *workflow decay*.

In this paper we present a novel framework that addresses workflow decay through the integration of system description, version control, container management and automated deployment techniques. It then introduces a set of performance optimization techniques that significantly reduce the runtime overheads caused by making workflows reproducible. The resulting system significantly improves the performance, repeatability and also the ability to share and re-use workflows by combining a method to uniquely identify task and workflow images with an automated image capture facility and a multi-level cache.

The system is evaluated through an extensive set of experiments that validate the approach and highlight the key benefits of the proposed optimisations. This includes methods for reducing the runtime of workflows by up to an order of magnitude in cases where they are enacted concurrently on the same host VM and in different Clouds, and where they share tasks.

Keywords: Provisioning Optimization, Workflow Reproducibility, Workflow Deployment, Container-based Virtualization, Cloud Computing

1. Introduction

Scientific workflows have become an increasingly popular paradigm for enabling and accelerating data analyses [1, 2]. They have been run successfully on many different computing environments including local PC, HPC clusters and the Cloud. In the Cloud workflows can exploit the economic and technical benefits including access to virtually infinite computing resources, and the pay-as-you-go charging model [3]. Yet, to maximise the benefits they provide, and facilitate the sharing of knowledge about the experimental methods they capture, workflows need to be reproducible. Reproducibility enables effective sharing as scientists can re-execute experiments developed by others and create new and/or improved experiments more

[☆]This document is the preprint version of the article available at: https://www.sciencedirect.com/science/article/pii/S0167739X18314377.

[*]Corresponding author
Email addresses: rawa.qasha@ieee.org (Rawaa Qasha), Zhenyu.Wen@ncl.ac.uk (Zhenyu Wen), Jacek.Cala@ncl.ac.uk (Jacek Cała), Paul.Watson@ncl.ac.uk (Paul Watson)

quickly [4].

Recent research on workflows and workflow management systems has found that a large number of workflows cannot be reused, nor can they produce the same results over time [5]. This has been termed *workflow decay* [6] and stems from a variety of factors, including: the lack of an adequate workflow description, missing resources required to execute workflows such as data and services, and changes in the workflow execution environment [7].

Work has been carried out to address the decay and enable effective sharing of workflows, their description and components [5, 8]. Some systems, like Galaxy [9] and ReproZip [10], rely on *physical preservation*, i.e. packaging workflows and their components into reusable modules and sharing these packages with prospective users. Other, such as Pegasus [11] and Taverna [12], focus on *logical preservation* which involves creating and sharing an abstract description of the workflow and its tasks, e.g. via web or in the form of Research Objects [7]. Both preservation techniques, however, rely on users to manually select, install and configure all the shared tools required to enact the workflows of their interest. Often, this is beyond users' capabilities or, at the very least, requires significant effort [13].

To address this problem, we have developed a new framework for provisioning workflows, their components and dependencies along with methods to automate this process. Our earlier work presented in [14], enables the automated provisioning of workflows, and uniquely combined both logical and physical preservation techniques. We use the Topology and Orchestration Specification for Cloud Applications (TOSCA) [15] to enable logical preservation through describing the workflows in a standardised way [16]; TOSCA defines a model for portable distributed Cloud applications, which helps run our workflows on a laptop PC and on different Clouds platforms. We then use container virtualization[1] to package workflows and their components so that they could be physically preserved and dynamically deployed on the Cloud.

Together, these two techniques allow users to customise workflows and tasks at the level of their description, which makes workflow development easier in practice. They also enable users to choose between more efficient single-container and more flexible multi-container execution mode. All that is combined with version control systems to provide automated management of changes in the source code and task images. Our framework offers ready-to-run workflows and workflow components, and addresses the majority of issues related to workflow decay and reproducibility.

However, the delivery of highly reproducible workflows comes at the price of additional overheads. We observed a number of performance issues that impact the packaging of workflow components, their provisioning and, ultimately, the overall effectiveness of workflow enactment. Specifically, the deployment of a workflow can become slow if task images are pulled from remote repositories too often, or if the same task or dependency is provisioned repeatedly.↩

Such situations occur quite commonly in practice as many scientific analyses require a re-execution of the same or similar workflows and tasks over time.[2] This is, for example, the case in Next Generation Sequencing in which tens, hundreds or even thousands of patients are screened for genetic variations using the same variant discovery pipeline.[3] Similarly, when running a parameter sweep analyses, the same workflow is executed many times with only some input parameters changed. It is also not uncommon that, in early stages of scientific analysis many variations of the same workflow are run before the scientist can confirm their hypothesis [17].

These inefficiencies were a barrier, preventing users benefiting from the advantages of our approach. For these reasons, in this paper we present optimisation techniques

[1] `http://www.docker.com`

[2] `http://www.recomp.org.uk`

[3] `https://www.genomicsengland.co.uk/the-100000-genomes-project/`

that overcome these problems by facilitating workflow and task re-usability, enabling their effective provisioning and also improving the sharing of scientific workflows and their components. We introduce image and cache management mechanisms which can greatly improve the performance of the provisioning and the enactment of our reproducible workflows. We also integrate these mechanisms with source version control, and so backward-compatible changes to the task code can be distributed transparently and on-demand across workflow enactment engines, while still maintaining workflow reproducibility.

In summary, the main contribution of this paper is:

- a full description of our framework for scientific workflow reproducibility including: workflow modelling, dynamic deployment, image management, and version control,
- new performance optimization techniques that enhance our reproducibility framework, including:
  - a new algorithm to name, create and select a compatible task image that improves the reusability of ready-to-run workflow components,
  - a multi-level cache of deployable components that supports workflow sharing and optimizes the workflow deployment process,
  - a cache of task artifacts and dependency packages to support the process of image creation,
- a set of experiments that use real and synthetic scientific workflows running on local and Cloud environments to validate and evaluate the proposed mechanisms. These show the improved performance of workflow provisioning and enactment, and improvements in workflow sharing.

The rest of the paper is organized as follows. In section 2, we give an overview of our framework, and then discuss our proposed optimization mechanisms in section 3. Next, the experiments and the experimental results are discussed. Before the conclusion, we review the related work in section 5

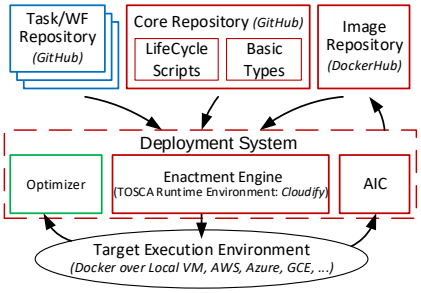

Figure 1: The architecture of our workflow reproducibility framework.

2. Framework Overview

Our framework [14], consists of six main components split into two layers (Fig. 1). The upper, repository layer includes the Task, Workflow, Core and Image Repositories, while the lower, deployment layer includes the Enactment Engine, Automatic Image Creation facility (AIC) and the newly added Optimizer. The deployment layer also directly communicates with the execution environment such as the Cloud or Docker infrastructure.

In brief, the Core Repository includes base types of workflow components and life-cycle scripts to manage them. The Task and Workflow Repositories are used to store and version the user-defined workflow and task descriptors that are specified using TOSCA.Importantly, the Task, Workflow and Core Repositories are backed by a version control platform such as GitHub. This provides the ability to track the complete history of the workflow and task changes, and allows the framework to control the changes that could potentially affect workflow's reproducibility; for more details please see [14].

The Image Repository contains workflow and task images automatically created by the AIC facility during workflow deployment. The primary goal of the AIC and Image Repository is to implement the physical preservation of workflows and tasks. The repository is also backed by

a version control system, in our case Docker Hub, which greatly helps in the building and management of image libraries.

The workflow Enactment Engine is a TOSCA-compliant runtime environment (e.g. Cloudify[4]) which can interpret and execute workflow descriptors written in TOSCA. Finally, the Optimizer has two main functions. One is the automatic selection of compatible tasks for the given workflow specification. The other is the automatic caching and sharing of workflow components to optimize the workflow provisioning. Both these functions are discussed in more details in Sec. 3.

Users interact with the framework primarily by contributing to the Task and Workflow Repositories (highlighted in blue) and by enacting workflows. The other framework components (highlighted in red and green) are used and maintained by a local system administrator.

2.1. Workflow Modeling

Our framework uses TOSCA to describe the entire structure of a scientific workflow together with all its components and details of the execution environment [16]. TOSCA is an OASIS specification for modeling a complete application stack, and automating its deployment and management in the Cloud [15].

The core of TOSCA modeling is the ServiceTemplate which consists of three logical parts: Node- and RelationshipType, TopologyTemplate and ManagementPlan [18]. The main intent of TOSCA is to improve the portability of Cloud applications in the face of the growing diversity of Cloud environments [19]. Thus, by exploiting TOSCA, we can turn a workflow into a reusable Cloud application that includes not only the description of a scientific experiment but also all details needed to deploy and execute it automatically.

Building a workflow using TOSCA starts by defining Node- and RelationshipTypes. A NodeType declares properties and life-cycle management operations of a workflow component such as a task, dependency and host. The properties include the component name, version and a URL to task artifacts, as well as its configuration parameters. The life-cycle operations include scripts that implement the deployment actions of workflow tasks, e.g. to create, configure and activate a task.

A RelationshipType can define two types of dependencies between nodes. A horizontal dependency imposes the desired data dependency between workflow tasks, whereas a vertical host-hosted relationship is added between workflow components and their host containers. Additionally, the RelationshipType can define actions required to materialise a particular relationship between components, e.g. how two tasks communicate or how a container can host a workflow task.

Using the defined types, the structure of the workflow is described as a graph of Node- and RelationshipTemplates embedded within a TopologyTemplate. The templates represent specific instances of the corresponding types. They provide values for the properties and implement the life-cycle operations declared in the types.

Then, the TopologyTemplate combines them into a workflow. Importantly, however, the TopologyTemplate includes not only the high-level structure of the workflow (i.e. task dependencies) but also all library dependencies and the definitions of containers and virtual machines that are supposed to host the workflow components. Thus, we can capture the complete software stack required to deploy and enact the workflow such as one presented later in Sec. 4.

2.2. Workflow and Task Repositories

An advantage if our framework is that the Workflow and Task Repositories can exploit, well used, publicly available platforms like GitHub. Thus, users can continue to use their favored repository for workflow and task development.

[4] `https://cloudify.co`

Repository structure. A Workflow/Task Repository aggregates various artifacts with resources required for its automated provisioning. Additionally, it includes the information needed by users so they can better understand the purpose of the workflow or task. Each repository consists of at least: TOSCA-based descriptors, workflow/task specific life-cycle scripts, sample data, a human readable description, *one-click* deployment scripts and deployment instructions.

Among them, the key elements are the TOSCA-based descriptors. In the case of a workflow, it is described as a single TopologyTemplate (as covered earlier). In the case of a task, two descriptors are needed. First is a NodeType that defines the task interface and refers to the actual task implementation code. Second is a TopologyTemplate of a test workflow which includes a sample NodeTemplate to illustrate how to use the task.

Other artifacts are important to maintain reproducibility. For example, provided with sample data and the one-click deployment script, users can easily test a workflow or task in their own environment. The script starts a multi-step process which deploys the workflow together with basic dependencies such as Docker and Cloudify and then enacts it. Another element of the repository, which can help users to understand the purpose of the workflow or task, is the human readable description. It includes information about the workflow function, inputs, outputs, tools required to deploy it, and the specification of the execution environment (for details see our sample repositories[5]).

Change control. One of the major reasons of workflow decay are changes in the workflow components. In a living system they are inevitable because the components (tasks, libraries and other dependency workflows) undergo continuous development. Thus, to maintain reproducibility we need to control them such that they do not contribute to the decay.

[5] `https://github.com/WF-ShAre`

Our approach maintains each workflow and workflow task in a separate code repository [14]. This brings multiple benefits: the repositories mark clear boundaries between components, offer independent version control and allow for easy referencing and sharing. Additionally, they provide branches and tags to implement the strict control of workflow and task interface.

Note that changes that occur naturally during workflow and task development can affect two layers: the interface and/or implementation of a component. Changes in the interface, such as adding a new input parameter, usually indicate some important modification to a component, and need to be followed by changes in its implementation. Conversely, changes to the implementation, if made with backward compatibility in mind, are often merely improvements in the code which can remain unnoticed. By keeping each component in a separate repository, we can control these two types of changes effectively. We use branching to denote changes in the interface, and tagging to indicate significant improvements in the implementation.

2.3. The Core Repository

The Core Repository is the fundamental part of our framework. It includes two types of components which shape how workflows and tasks are connected and managed. Firstly, the Core contains the descriptors of base Node- and RelationshipType that define the abstract workflow task, library dependency, and link to connect the tasks. For example, all workflow task types must be derived from the base workflow_service type defined in the repository.

Secondly, the Core includes the life-cycle scripts that implement common operations related to the management and enactment of workflows and tasks. They are responsible, for example, for task initialization, data preparation and data transfer between tasks, but also for the creation of a task/workflow Docker container.

Using the types and scripts defined in the Core Repos-

itory, developers can implement tasks and workflows that are compatible with our framework and which can be automatically managed and enacted.

2.4. Transparent Image Management

Together with improving reproducibility, the main goal of our framework is to effectively share workflows and tasks. We want them to be ready-to-run components that can be automatically deployed and easily re-used to facilitate the rapid development of new experiments.

The basis for seamless deployment is, however, effective provisioning in which image management plays a crucial role. Although TOSCA-based descriptors alone are enough for our framework to automatically deploy and enact workflows, using just them would incur significant runtime overhead. The framework would repeat the same, sometimes long running steps to deploy a task every time it is executed.

To minimise the overheads we have therefore designed and implemented our framework such that it can effectively use Docker images. The images may encapsulate some or all of the deployment steps, and so speed up the enactment phase. Specifically, the framework can use a pure-OS image commonly available from Docker Hub or a specialized user-defined image which includes some workflow/task dependencies, or even a complete image that contains all of the required dependencies. If the image referred to in the workflow TopologyTemplate does not contain all the dependencies, they will be installed by the framework on-demand during workflow enactment. That automation simplifies the development cycle because developers are not forced to manually prepare and manage task or workflow images before sharing a workflow.

However, to further simplify the use of the framework we have designed and implemented an Automatic Image Capture component. Using the Docker image manipulation operations, the AIC is able to create workflow and task images automatically from the container used to provision a task. These images are then deposited in a private or public Image Repository and re-used next time a workflow or task is executed. The automation of this process is based on the image management technique proposed in this paper makes the framework very flexible - the user may start with any Docker image they prefer and, in the end, the framework will make sure that the task or workflow deployment steps are captured and are not repeated needlessly. As shown later in the Evaluation section, this simplification can have an extremely positive impact on the runtime performance.

2.5. Deployment Configuration Options

To further enhance the flexibility of our framework it supports also two deployment configuration options: *single-container* to deploy the whole workflow at once for the shortest enactment time, and *multi-container* for isolated deployment of workflow tasks and the higher re-usability of their code and images. The configurations influence the way in which the deployment and enactment of workflows is performed, and also determine which type of image the AIC will create for the workflow.

In the single-container configuration all tasks and their dependencies are provisioned in one container and a single image corresponding to the whole workflow is created. Fig. 2 depicts the detailed steps of workflow deployment in that case. The process starts with initial enactment preparation step followed by pulling a workflow base image specified by the user, e.g. a generic pure-OS image. Then, using the image a Docker container is created. The next step is the installation and configuration of the dependencies required by all tasks, which involves package downloading and on-line installation. Next, each task artifact is downloaded from the Task Repository followed by input data retrieval and task execution. The operations of task download, data retrieval and task execution are repeated for all tasks in the order specified in the TopologyTemplate. Finally, once all tasks have completed, the

workflow image is created and cached, and the workflow container is destroyed.

The multi-container deployment scenario is depicted in Fig. 3. Again, the process starts with the initial enactment preparation which, in this case, is followed by the task enactment loop repeated for each task in the workflow. The task enactment starts by pulling the base image for the first task in the workflow. Using the image a Docker container for the task is created. This is followed by the installation and configuration of task dependencies. Next, the task artifact is downloaded from its repository and the task image is created. Then, the framework retrieves the task input data and executes the task. Finally, the container used to execute the task is destroyed.

Both configurations have their advantages: the former imposes less overhead in terms of storage and performance, whereas the latter promotes better reuse of task images and gives more flexibility in tracking task changes, which is useful when the workflow requires updates. And both, single- and multi-container configurations can equally well support the repeatability and reproducibility of workflows.

3. Optimization of Workflow Deployment

As mentioned earlier, without a proper optimization approach the enactment of portable workflows may become ineffective due to repeated task provisioning actions. Thus, the aim of the new Optimizer component is to facilitate the selection of a suitable image to provision a task, automate the sharing and re-usability of workflows, and optimise the overall workflow provisioning process.

The Optimizer tackles situations when tasks or other workflow components are used in repeated invocations of one or more workflows. Importantly, it does not require any change to the structure of workflows and tasks we developed earlier. We now describe how this is achieved.

3.1. Just-in-time Image Naming and Selection

Workflow tasks in our framework are deployable components, and so the task developer must specify the base image id (image name and version) which they want to use to execute the task. Usually, it is an id of a pure-OS image taken directly from the Docker Hub, but it may also be some specific image that the developer prefers to use instead.

To create an effective way to choose an appropriate image used to provision a workflow task, we automated the process of naming and selecting images that can be used to create a task container. We use a simple yet robust naming convention that draws on naming practices recommended by the version control platforms we use to manage source code and images - Git and Docker.

Given the task's base image id we construct the *task image id* as base_image_id.task_name.task_version. In this way we link within a single identifier three relevant elements that influence task provisioning: (1) the base image id is important as it will be used to provision the task if no appropriate task image exists, (2) the task name uniquely identifies the repository of the task included in the workflow, (3) the task version refers to a specific tag in the task's code repository. The combination of these three elements makes the task image id unique and is important for its reproducibility. For example, if a new task version is released, it may be tagged so a new task image will be created with a new id. Then, workflows that rely on the previous task tag will still use the old id, and so keep using the previous image for the task.

Importantly, when the workflow developer includes a task in their TopologyTemplate, they can choose to specify the task's branch name rather than a specific tag. In this case, the framework will automatically detect the latest tag of that branch and use it as the task_version part of the task image id. This gives workflows the ability to automatically track minor backward compatible updates and improvements made by task developers.

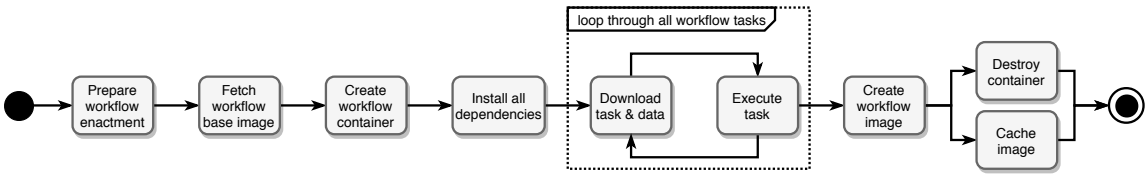

Figure 2: Deployment steps in the single-container configuration.

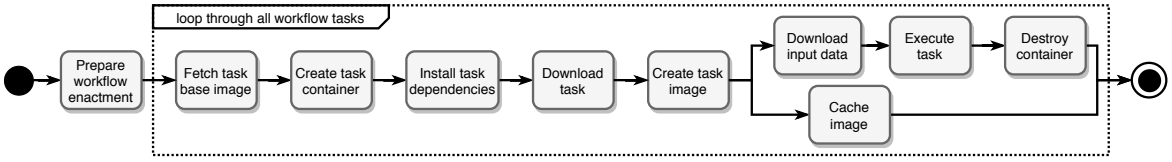

Figure 3: Deployment steps in the multi-container configuration.

Clearly, the proposed naming scheme requires consistent tagging and branching of the code; more details about it can be found in [14]. The developers who would like to create tasks and workflows compatible with our framework must adhere to it. Using the proposed naming scheme we can, however, implement an effective image selection algorithm outlined in Alg. 1.

The algorithm iterates over each task in a workflow, identifies the image used to provision the task and makes sure that the image is available in the host execution environment. In line 7 the *task_version* is determined. As mentioned above, it is the appropriate tag from the task repository. Then, following the discussed naming pattern, in line 8 the task image id is set. Based on that identifier, the search process takes place to find the image in the three-level cache (lines 9–16). The search process starts by looking for the task image in the host environment. If that fails, it moves to the second-level cache which is a local repository that can be accessed by authorized users. Again, if the target image is not found at that cache level, the search will proceed to the third level – a public repository in Docker Hub. Finally, if no task specific image is found, the task's base image will be used (lines 18–19).

Note that given the algorithm and unless the code of a task changes, the same task image is used to execute the task in all workflows in which it is included. This greatly improves the effectiveness of provisioning and the performance of workflow enactment, especially if the same task is executed multiple times by one or more workflows.

And for the single-container configuration we use a very similar approach to image naming and selection. Then, instead of looking for a task image the system tries to locate a workflow image following pattern base_image.workflow_name.workflow_version.

3.2. Image Caching and Automatic Sharing

Once the enactment of a workflow ends, all the task images are available in the host execution environment for shared with others. Although sharing of workflows and their components is supported by a few workflow management system such as Pegasus, Taverna and Galaxy our approach is novel - it not only allows the structure and description of workflows and tasks to be shared but also provides a portable way to deploy and execute a workflow. Users of our workflows and tasks can easily run them in an environment of their own choosing (e.g. local PC, local server or cloud) by means of a one-click deployment script.

A major, additional benefit is that workflow developers can combine these ready-to-run tasks, without incurring the burden of provisioning the software stack for each task. Further, the framework also automates the process of publishing the workflow and task images, making them

Algorithm 1: Just-in-time task image selection.

```
1  ST_W – ServiceTemplate of workflow W;
2  NT_t – NodeTemplate of task t ∈ W;
3  I – a map of {t → img_id} s.t. image img_id is used
     to create task t;
4  for NT_t in ST_W do
5      base_image_id ← get the id from NT_t;
6      task_name ← get the name from NT_t;
7      task_version ← get the tag of t or its branch
         from GitHub;
8      img_id ←
         base_image_id.task_name.task_version;
9      if image img_id in the host cache then
10         I[t] ← img_id;
11     else if image img_id in the local cache then
12         copy image to the host environment;
13         I[t] ← img_id;
14     else if image img_id in the Docker Hub then
15         docker pull img_id;
16         I[t] ← img_id;
17     else
18         docker pull base_image_id;
19         I[t] ← base_image_id;
20     end
21 end
```

immediately available both locally and in a public repository. As a consequence we can:

- minimise the effort required by the user to re-execute a workflow, and
- significantly reduce the overall workflow deployment time
- significantly reduce the overall workflow enactment time

The foundation of image sharing in our framework is the three-level cache, shown in Fig. 4. It supports the following use cases:

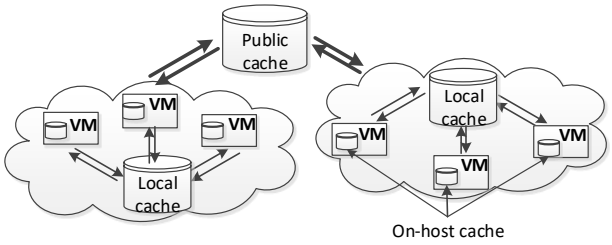

Figure 4: Three-level cache for task and workflow images.

- The first level cache in the host environment enables the fastest workflow deployment. If all the required images are available in the host, there is no need to download and install any workflows and tasks, as they can be directly provisioned by Docker.
- The second level – local repository – enables a controlled way of sharing private images within organizational boundaries, and also supports off-line deployment when there is no access to the Internet/public repository.
- The third level – public repository – supports the sharing and re-use of images across organizations and facilitates workflow execution in different Clouds platforms.

We implemented the cache facility on top of the standard tools provided by the Docker platform. The first level cache is the on-host Docker image repository. The second level cache is an instance of the Docker image repository running on a dedicated machine within an organisation. The third level, public repository is realized as the Docker Hub public organization.

3.3. Caching Workflow Component Artifacts

The techniques presented above support performance optimization for the provisioning process when tasks/workflow images are available. However, we designed and implemented an additional optimization mechanism when the workflow is deployed for the first time, and to provide support for the development of the new task images. It is a

cache of artifacts, such as task code files, dependency tools and libraries that are essential in building a task and/or workflow image.

A common pattern in scientific workflows is that tasks within one workflow or a family of workflows share common libraries and dependencies. Given our proposed image naming convention, each of these tasks is packaged in a separate image and follows a separate image creation cycle. This means tasks cannot share dependencies. Although isolated task provisioning is useful for the various reasons (e.g. easier image reuse and avoiding dependency versioning issues), it may cause some redundant network traffic.

Therefore, to minimise the time required to create similar images we also cache the workflow component artifacts. In this way we reduce the time needed to create the images, and so positively influence the performance of workflow enactment. The cache also plays a very useful role for developers as they usually test many more task and workflow versions before they can tag an official release that generates a task image.

3.4. Optimized provisioning and enactment

We combined the naming, selection and caching mechanisms described above and embedded them into our framework to improve the deployment and enactment of workflows. Fig. 5 depicts the detailed steps our framework follows during task provisioning; note that similar steps are implemented if the single-container configuration is used.

In the first step Alg. 1 is run to determine which image is going to be used. If the task image is available, the framework can create a container and immediately proceed to task execution. Otherwise, if the task's base image is used, the framework creates a *base container* and then runs through the TopologyTemplate to deploy task dependencies and artifacts. An image is created and cached under the unique task image just before task execution, for the benefit of any future provisioning requests for that task. The task image is created immediately after downloading task dependencies and artifacts, therefore, no task input, output or intermediate data is included, and so the image can be safely used to repeatedly re-execute the task.

Although the process of image creation has some influence on the workflow deployment time, we can save significant amount of time by re-using these images to provision the same task in future executions. In the evaluation section we present the benefits and overheads of the proposed strategy.

4. Experiments and Evaluation

To evaluate the performance of our framework and the proposed optimization techniques we conducted a series of experiments on a number of real scientific workflow applications. The aim was to investigate the effectiveness of our framework from various angles including the performance of workflow deployment, re-use and sharing, and storage requirements.

4.1. Experimental Setup

We ran the experiments on a number of real and synthetic workflows which differ in size, structure and functionality. The workflows we selected are: *Neighbor Joining* (NJ; 11), *Sequence Cleaning* (SC; 8), *Column Invert* (CI; 7), *File Zip* (FZ; 3), *WF-1* (6) and *WF-2* (8); shown in brackets is the number of tasks. As an example, Fig. 6 depicts the structure of the *SC* workflow used in NGS pipeline [20]. The structure of other workflows is presented in Fig. A.15 in Appendix A.

NJ and *SC* were originally designed in e-Science Central [21] and reimplemented using our framework. Other workflows were created specifically to be used in the experiments. The workflows cover a set of interesting features including: 1) non-trivial structure; for example *NJ* and *SC* realize the sequence, and split and merge workflow patterns, 2) different number of tasks, from 3 to 11, 3) variety of dependency libraries and tools required to execute tasks,

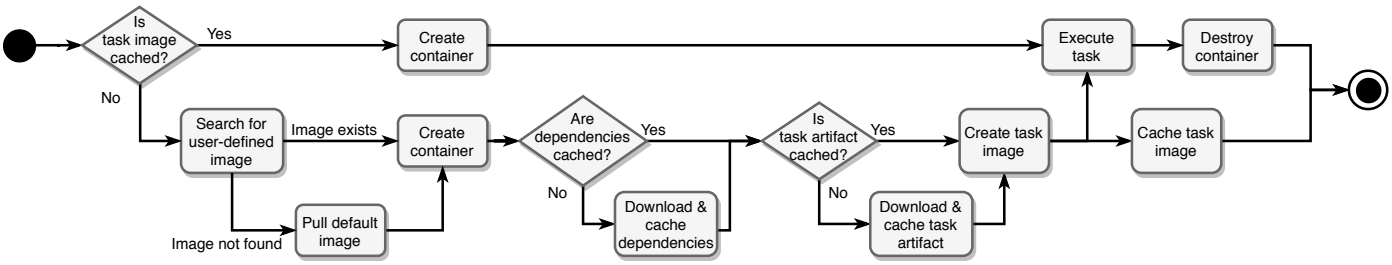

Figure 5: Provisioning steps for a workflow task with automatic image selection and caching.

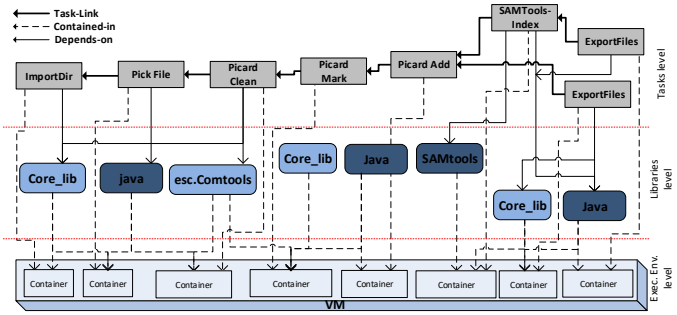

Figure 6: The structure of the *SC* workflow in multi-container configuration described in TOSCA.

e.g. *NJ* uses the Wine library[6] that needs relatively large time and disk space to be installed; some workflows and tasks share also a set common dependencies, 4) common tasks to evaluate the effectiveness of sharing optimisation, e.g. *NJ*, *WF-1* and *WF-2* have a few tasks in common.

4.2. Workflow Repeatability on Different Clouds

To illustrate the potential of our framework in supporting repeatability and reproducibility and the value of the proposed workflow representation we conducted the following experiment. A workflow, initially designed and created in a local development environment, was re-enacted on a local VM and three different Clouds: Amazon AWS, Google Engine and Microsoft Azure; the configuration of the VMs is presented in Tab. 1. In the experiment we used four workflows: *Neighbor Joining* (NJ), *Sequence Cleaning* (SC), *Column Invert* (CI) and *File Zip* (FZ). Initially, we recorded the execution time of the first enactment in the development environment, which also automatically created one or more Docker images. Then, we re-executed workflows five times in each VM and collected results.

[6] `https://www.winehq.org`

Table 1: Basic details about the execution environments.

Environment	CPU cores	RAM [GiB]	Disk space [GB]	Operating system
Amazon EC2	1	1	8	Ubuntu Srv 14.04
Google Cloud	1	3.75	10	Ubuntu Srv 14.04
Microsoft Azure	1	3.5	7	Ubuntu Srv 14.04
Local VM	1	3	13	Ubuntu 14.04

Each workflow was run in two different configurations: single- and multi-container to show the overheads of the latter. The workflows were deterministic, and so the output data obtained were exactly the same irrespective of the configuration and execution environment used. The average execution times were also similar although longer for the multi-container configuration. Fig. 7 shows a chart with the results for the SC workflow, while Tab. 2 includes the results for the other workflows.

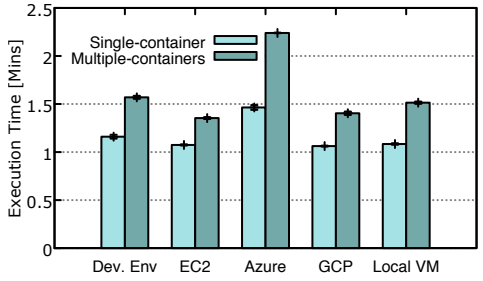

Figure 7: The average execution time for the *SC* workflow executed in different environments.

The experimental results show a number of interesting

Table 2: The average execution time and standard error of the mean (in minutes) for the selected workflows executed in different environments.

	Neighbour Join.		Column Invert		File Zip	
	Single	Multi	Single	Multi	Single	Multi
Dev. Env.	2.13 (0.11)	2.54 (0.02)	0.90 (0.09)	1.30 (0.02)	0.60 (0.01)	0.94 (0.04)
Amazon	1.74 (0.14)	2.27 (0.03)	0.66 (0.09)	1.18 (0.01)	0.50 (0.01)	0.84 (0.11)
Azure	2.52 (0.01)	3.86 (0.16)	1.35 (0.01)	2.10 (0.02)	1.23 (0.03)	1.38 (0.02)
Google	1.52 (0.01)	2.48 (0.01)	0.74 (0.1)	1.18 (0.01)	0.50 (0.02)	1.01 (0.01)
Local VM	1.65 (0.09)	2.50 (0.07)	1.03 (0.01)	1.37 (0.01)	0.53 (0.01)	1.03 (0.01)

aspects. First, our scientific workflows are repeatable – they can run in different environments and produce the same outputs with similar runtime. Second, the execution using the multi-container configuration took longer than for a single-container. This result was consistent across all environments and stems mainly from the fact that in the multi-container mode dependencies common across tasks had to be re-installed multiple times, and also for each task a separate image had to be created. Third, the execution in Azure took longer than in other Clouds because the VM in Azure had smaller network bandwidth than VMs in other Clouds; a factor out of our control.

Additionally, the results demonstrate a common workflow development pattern supported by our framework. First developers build and test a workflow in their local development environment. Then, once the workflow is ready, they share it with users via Workflow, Task and Image Repositories.

4.3. Automatic Image Capture to Improve Performance

As described above, our framework is flexible enough to allow tasks and workflows to use pure-OS images available from Docker Hub or custom, predefined task/workflow images created by users or the AIC. By using a predefined image, the framework can avoid the installation of dependency libraries and task artifacts required during workflow execution. The elimination of some of the deployment steps can have very positive impact on the runtime

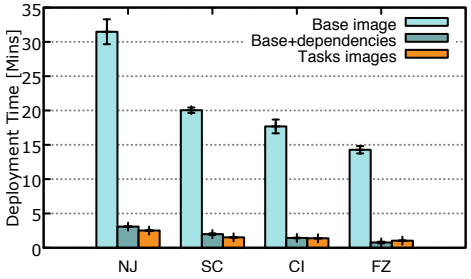

Figure 8: The average execution time of test workflows using different task images.

of workflows, and we prepared a set of experiments to illustrate it. We ran our workflows using different images: the base image available on Docker Hub, the base image with pre-installed dependency libraries, and task images captured by the AIC. Fig. 8 shows the average workflow execution time for four tested workflows.

Clearly, there was a significant overhead in using the base image from Docker Hub. The main reason was the time required to install dependency libraries such as the Java Runtime Environment (JRE) or, in the case of the *NJ* workflow, the Wine library. The second and third option (Base +dependencies and Task images) took similar time to complete with slightly shorter execution time for experiments which used task images created by the AIC. This is because the AIC captures everything the task needs to run (according to the task's TOSCA descriptor), whereas the second option included only dependency libraries, whilst the task artifacts were downloaded and installed on-demand.

The results explicitly show that the use of pre-packaged images is the fastest option. However, from the user perspective the quickest and easiest is to use base images already available on Docker Hub instead of building images manually. Our framework supports both these options at the cost of an overhead incurred by the initial execution of a workflow - the first run will follow the complete deployment cycle and create task images, whereas all subsequent executions will benefit from those images and run much faster.

4.4. The cost of Automatic Image Capture

As presented above, the proposed process of automated image creation provides various benefits to the user. It improves reproducibility, can streamline image building and helps optimise workflow provisioning. These benefits come do however come with an additional price – the space required to store the task and workflow images and containers.

To illustrate the amount of storage space needed we listed in Tab. 3 the images created during the enactment of the *NJ* workflow in the multi- and single-container configuration. The workflow includes 7 distinct tasks (11 altogether) and the table shows the size of their compressed image, as available from Docker Hub, together with the size of the uncompressed image and its top layer.

Images created by our framework consist of two parts. One is the top layer built by the AIC from the container used to provision the task. The other includes all underlying layers captured by the image provider during the creation of the base image. Thus, the bottom layers will be shared by all tasks that use this base image, whereas the top layer is unique to each task and will only be reused by multiple invocations of that task.

As shown in the table, the image sizes of our tasks may become substantial if based on an image too lean in relation to the actual task dependencies. For example, images based on pure-OS ubuntu:14.04 compressed take over 220 MB, whilst their uncompressed top layer requires about 300 MB of space. The main reason why they need so much storage is because all of them depend on the JRE which is installed on top of the base image. In this a case, i.e. when multiple tasks depend on common libraries, it is better to indicate another, more specialised base image.

The bottom part of the table includes a selection of the task images built using the openjdk:7-jdk base image. As shown, for most of the *NJ* tasks the AIC adds to the base image a top layer of only a few MB – the size of the task artifact. The exception is the MegaNJ task which requires Wine, a relatively large software library.

Interestingly, as more and more diverse tasks are added to the workflow, the developer may be tempted to create a dedicated task image which is going to include all dependency libraries needed by all the tasks in a workflow. This approach, however, has two clear disadvantages. Firstly, it creates a large image which over time is difficult to maintain without significant overheads. Secondly, it may impose even higher requirements on the storage.

In Fig. 9 we show the size of the Docker on-host cache during the enactment of the *NJ* workflow and in relation to the selected base image. The nj-specialized is the image which includes all task dependencies for that workflow, the openjdk:7-jdk is the official Docker image providing JRE, whilst ubuntu:14.04 is the official pure-OS image with Ubuntu Trusty Server. As shown in the figure, the most efficient with regards to cache size is the openjdk:7-jdk image. It needs the smallest total storage space of 5.35GB, whereas the other two base images require nearly twice as much. Also, the nj-specialized base image generates overhead during the creation of containers (cf. the vertical dotted lines). It is because each container is started with the complete set of relatively large libraries.

Surprisingly, the use of the pure-OS image is also not very space efficient, especially at the end of the workflow when four task containers are active. All of them with freshly installed JRE consume significant amount of

Table 3: The size (in MB) of task and workflow images of the *NJ* workflow created using AIC.

Image name	Image type	Compressed size	Uncompr. size	Top layer size
dtdwd/ubuntu-14.04_importfile-task.v1.0	task	220	496	308
dtdwd/ubuntu-14.04_file-join-task.v1.0	task	222	498	310
dtdwd/ubuntu-14.04_filter-duplicate-task.v1.0	task	222	498	310
dtdwd/ubuntu-14.04_csv-export-task.v1.0	task	220	496	308
dtdwd/ubuntu-14.04_exportfiles-task.v1.0	task	221	497	309
dtdwd/ubuntu-14.04_mega-nj-task.v1.0	task	583	1476	1288
dtdwd/ubuntu-14.04_clustalw-task.v1.0	task	227	508	320
dtdwd/ubuntu_14.04-nj-1container-fullcaching-caching	workflow	611	1535	1347
dtdwd/openjdk7-jdk_importfile-task.v1.0	task	215	474	1.5
dtdwd/openjdk7-jdk_filter-duplicate-task.v1.0	task	217	476	3.4
dtdwd/openjdk7-jdk_mega-nj-task.v1.0	task	419	985	513
dtdwd/openjdk7-jdk_clustalw-task.v1.0	task	222	486	13.4
dtdwd/openjdk7-jdk-nj-1container	workflow	483	1093	621

space.[7]

The presented storage requirements indicate that the on-host and local cache layers may need a cache eviction policy to work effectively (for a recent survey see, e.g. [22]). This may be especially useful for prolonged execution of our platform and when a diverse set of tasks and workflows is used. We have left the implementation of this aspect, however, for the future work.

4.5. Influence of Task Changes on Deployment Time

One of the challenges for a workflow management systems is the ability to apply changes to only parts of the workflow without affecting the remaining, unchanged elements. In our work, we have addressed this challenge by (1) isolating the provisioning of each task in a separate Docker container and (2) packaging the full stack of task software as a Docker image. This is supported by tracking task versions and using the version information to select the matching task image. When a new version of a task is detected, the framework searches for the relevant task image. If however no compatible image is available then, the full provisioning of the task takes place.

To show the validity of our approach in selecting task image and tracking changes we applied several changes to some tasks of the *NJ* workflow and ran an experiment that covers two cases. Firstly, an image for the changed task has already been created and so it was used to provision the task. In the second case no image existed for the changed task, thus to execute it the base image was used followed by the installation of dependencies. In this way we ensured that we used both scenarios: full task provisioning and reuse of an already-created task image.

The experiment, as shown in Fig. 10, starts with the full deployment of the *NJ* workflow (Full-depl), i.e. full provisioning of each task and creation of their images. This is followed by a few executions using task images available from the on-host cache (Cached-image). After several executions, a new version of **task1** was created, and so no

[7] A part of the space requirements is generated by the use of the recommended `apt-get cache update` command which makes the images significantly larger.

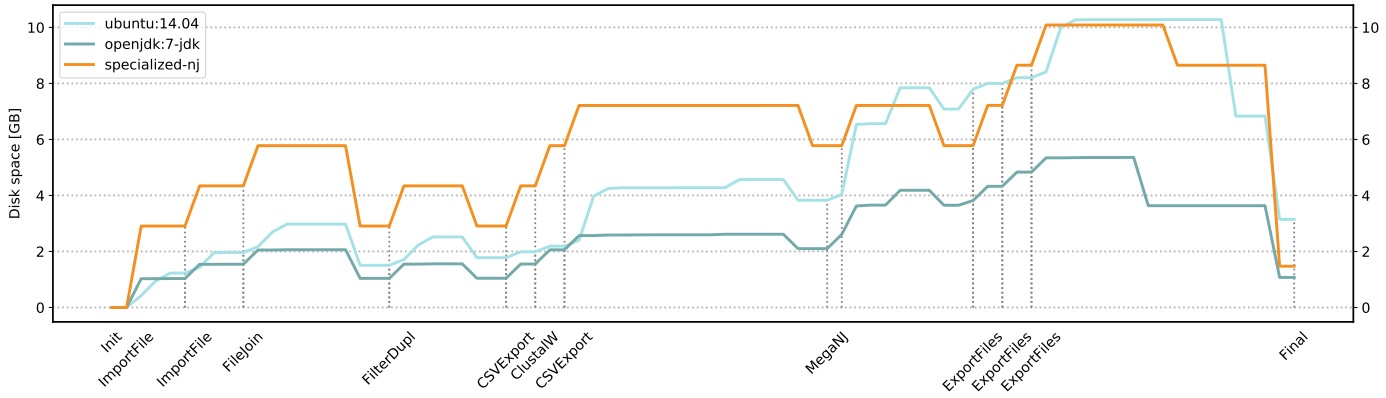

Figure 9: The size of on-host cache `/var/lib/docker` during the enactment of the *NJ* workflow using the multi-container configuration and three different base images.

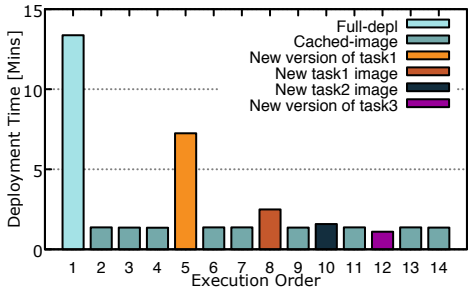

Figure 10: The impact of task changes on the workflow execution time.

compatible task image was found. In effect the new task version had to be fully provisioned. Again, after several executions with Cached-image, a new version of **task1** is discovered, but with a task image available in the public repository. While continuing to re-deploy the workflow, a new version of **task2** is recognized, and a corresponding task image was available and reused in this execution. Finally, a new version of **task3** was found with no related image, so again the full provisioning of that new task version had to be carried out.

The results show that the changes of any task do not much affect the execution of the workflow. The new version of the task is automatically started by either using a pre-created task image or by provisioning the full software stack for the task.

Furthermore, the just-in-time selection algorithm has been used in the experiments, which demonstrates that the algorithm is able to effectively select and re-use the new compatible task image whenever it becomes available. This improves the performance and reliability of workflow re-execution. Note that the full deployment of a task requires much more time when compared with the deployment of pre-created task images. This is because of the time required to install task dependencies, especially if they take a considerable amount of time, such as the first case of provisioning the new version of **task1**.

4.6. Task Image Caching for Deployment Optimization

In this experiment, we aim to show the effectiveness of the automatic creation of task images and the sharing and reuse of them through multi-level caching. This experiment consists of two parts. The first part shows the influence of creating task images on the overall deployment time, while the second part presents the advantage of multi-level caching and re-use of task images in different cases during the optimisation of workflow provisioning.

Tab. 4 presents the results of the first part. The *NJ* workflow has been executed in three different cases: the Cache-off/clear-cache deployment with caching switched off and an empty cache such that no image existed in any of the caches, the Cache-on/clear-cache deployment with caching switched on, automatic image creation and empty cache, and the No-image-creation deployment with cache

Table 4: The share of task image creation in the total workflow execution time.

Deployment case	Total execution time [s]	Image creation time [s]
Cache-off/clear-cache	1883.6	355.5 (18.9%)
Cache-on/clear-cache	967.0	276.5 (28.6%)
No-image-creation	505.2	–

switched on but the AIC switched off.

The results show that creating task images takes a relatively large amount of time, and substantially increases the initial deployment time. However, image creation takes place only once, when the workflow is deployed for the first time. Thus, the time spent on image creation will be repaid for all subsequent deployments and enactments as they will rely on the existing images instead of re-provisioning the full software stack of each task.

The results of the second part are shown in Fig. 11 which presents four scenarios for *NJ* deployment. Cache-on/clear-cache refers to workflow deployment with caching switched on and empty cache. While Cache-on/local-cache, Cache-on/public-cache and Cache-on/full-cache are deployment scenarios with caching switched on and task images available in a local repository, public cache, and local execution environment, respectively.

As shown in the figure, there is a considerable reduction in the execution time when task images are used instead of re-deploying all tasks. We observed the fastest deployment time when the task images were available locally in the execution environments. The overall execution time was reduced nearly 20 times from 1883.6s for the Cache-off/clear-cache case to only 96s for the Cache-on/full-cache case. In addition, there was a considerable reduction in the runtime for the Cache-on/local-cache and Cache-on/public-cache cases when compared with the full deployment and no cache case, i.e. from 1883.6s to 380.9s and 414.2s, respectively.

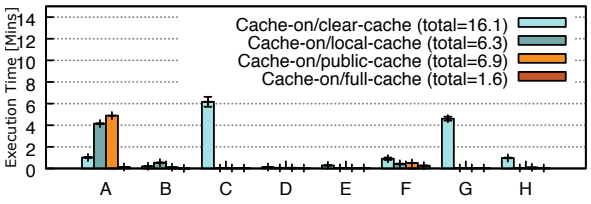

Figure 11: Execution time for the *NJ* workflow with task image caching: (A) image download, (B) container creation, (C) tools & libs inst., (D) data copying, (E) task download, (F) task execution, (G) images creation, (H) container removal.

4.7. Sharing Cached Images between Workflows

The previous experiments showed the effectiveness of the automatic creation, sharing, and re-use of task images for individual deployments of a workflow. In this experiment, we show the impact of using our optimisation techniques in the case of the concurrent deployment of two instances of the same workflow. The concurrent deployments were running either in the same execution environment or in different ones. The aim of this part of the experiment is to demonstrate the impact of sharing and re-using of task images between two workflows, which have some tasks in common. Here, prior to workflow deployment the image cache was emptied.

We have conducted this experiment for two scenarios: (1) concurrent executions of two instances of the same workflow (on the same and two distinct hosts) and (2) concurrent executions of different workflows on different Clouds.

4.7.1. Concurrent Executions of the Same Workflow

The main reason for automating the process of image sharing and selection is to make sure that the image is created once and can be immediately re-used in the same or another workflow. Therefore, users in different places can instantly use the images that are created by others.

To evaluate our approach in this situation we deployed two instances of the *NJ* workflow concurrently with a delay between the start of their execution. The experiments have been conducted with two different execution envi-

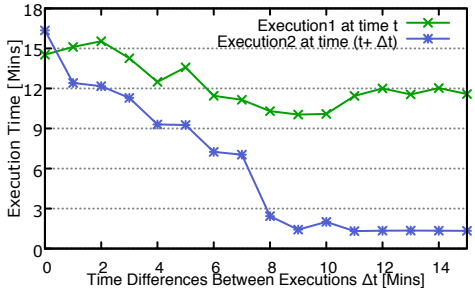

Figure 12: The Influence of sharing components between two instances of *NJ* workflow on a single machine.

ronments to show how the images can be shared during deployment time and the impact that the executions have on each other.

In the first case, the two instances of the *NJ* workflow are executed on a single machine. The case represents, for example, a very common scenario in which the user has a set of input samples to process, each by a separate instance of a workflow, but the workflow tasks are single-threaded and cannot exploit multiple cores at once. As shown in Fig. 12, if the two instances start with no delay, *Execution2* takes slightly longer than *Execution1* because *Execution1* seizes more shared resources. For the subsequent executions, we triggered the second instance after the first execution *with increments of one minute as time interval* Δt. For example, if the time difference between the two execution is 5 (the value on the x-axis), *Execution2* was started 5 minutes after the start of *Execution1*.

The figure shows an interesting result for both executions. There is a considerable decrease in the execution time, particularly for *Execution2* when the delay between the starting times increases. The reason is the sharing and re-use of workflow components (task artifacts, dependency packages, and task images) between the two instances. The first instance starts provisioning the tasks, which involves downloading and installing these components. The application of our optimisation techniques makes the components available for utilization by all subsequent executions. As the difference in the start times for the two instances is increased, greater benefits in the overall execution time are gained. The optimal time is reached when the second execution starts nine minutes after the first, i.e. after all task images have been created.

Another important result is the reduction of time for *Execution1*. This happened when *Execution2* reused some components prepared by *Execution1* and so was able to "overtake" *Execution1*, which was slowed down by the time needed to creat and push a task image. In effect, *Execution2* could start downloading and preparing a subsequent task, and so paved the way for *Execution1*.

Clearly, our optimization techniques enable sharing of different components between workflows running concurrently in the same environment. Additionally, when running multiple workflows on a single multi-core machine, they can reduce the deployment time and improve the runtime performance.

In the next experiment we tested another case of running two instances of *NJ* concurrently but on different environments (local machine and the Google Cloud Platform). Since the workflows are running in different environments, they can share task images only via the public repository (Docker Hub). This case is useful when two users decide to run an instance of the same workflow at the same time or with a small time difference. Consequently, the tasks images created by one of them can be used by the other which will avoid the need for full task provisioning.

As previously discussed, we ran the instances several times in parallel with time shifts starting from zero and then increased by intervals of one minute until they ran sequentially. To make the results comparable the cache was emptied prior to each execution.

The results in Fig. 13 show that the execution times are largely stable for the instance running on the Google Cloud Platform. This is because the instance is always started first and requires full task provisioning with no task image available. Whereas, the execution times for

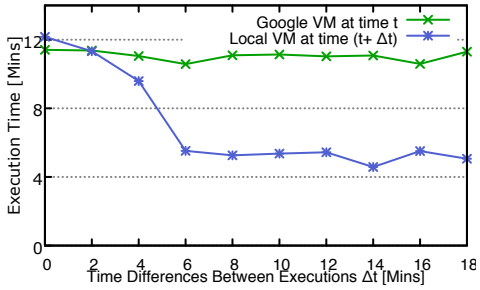

Figure 13: The influence of sharing tasks images between two instances of *NJ* workflow on different clouds.

the second instance running in the local machine decreases gradually as the time difference with the first instance increases. Again, the reason for this is the ability to re-use the task images created by the first instance. All created images are pushed immediately to the public repository to be shared and re-used by others.

Comparing the results shown in Fig. 12 and 13, clearly, the execution on a single machine was slower (by three minutes). Importantly, however, it indicates that when workflows share common tasks, running one can speed-up the execution of the other, irrespective of where the workflows run. We decided to look at that interesting behavior more closely.

4.7.2. Concurrent Executions of Different Workflows

In this part, we conducted an experiment similar to the previous one but with different workflows running in different Cloud environments. Our aim is to show how task images can be shared by workflows that have only a few tasks in common.

We have selected three different workflows: *NJ* which has four and five tasks in common with *WF-1* and *WF-2* respectively, while *WF-1* and *WF-2* has three tasks in common. In addition, *NJ* and *WF-1* were executed in two Google VMs, whereas *WF-2* was executed in our local machine. The workflows were executed in different order as shown in Fig. 14a–c. In each case, the workflows were run concurrently and their start time was shifted in intervals of one minute. For example, in Fig. 14a *WF-1* starts first, then, after one minute *WF-2* starts, while *NJ* starts execution one minute after the second workflow. In the next sequence interval Δt was increased by one minute, thus the start time delay became two and four, and so on. Again, the caches were empty before the start of each set of executions.

The results presented in the three figures show stability in the execution time of the first workflow (the green line), while there is a gradual decrease in the execution time for the other two. As previously explained, this is because executing the first workflow involved full provisioning for each task, creating task images and publishing them to the public repository, from where the others reused them. The actual reduction in runtime depends, however, on different factors such as the number of common tasks between the workflows, the time differences between the executions, and the order of the shared tasks in the workflow structure.

From the results shown in Fig. 14, we can see that there is a considerable decrease in the execution time of *NJ* and a slight reduction of runtime for *WF-2*. This is because *NJ* is the third running workflow which means it can reuse the task images created by the first two (up to nine images). While there are fewer tasks shared between *WF-2* and *WF-1*.

In general, the third workflow in each case gains the most benefit from re-using the images, with a reduction in the execution time proportional to the number of reused images created earlier. Also, *WF-2* was the quickest of the three workflows as it does not require any large dependency library, while both *NJ* and *WF-1* include a task that depends on the Wine library. We observed also some instability in the execution runtime, e.g. Fig. 14a shows an increase in the execution of *WF-2* and *NJ* when Δt was 2 and 4, respectively. We observed that this was due to differences in the network traffic at different time. The fluctuations affected the image upload and download times, to and from the public repository and thus the overall ex-

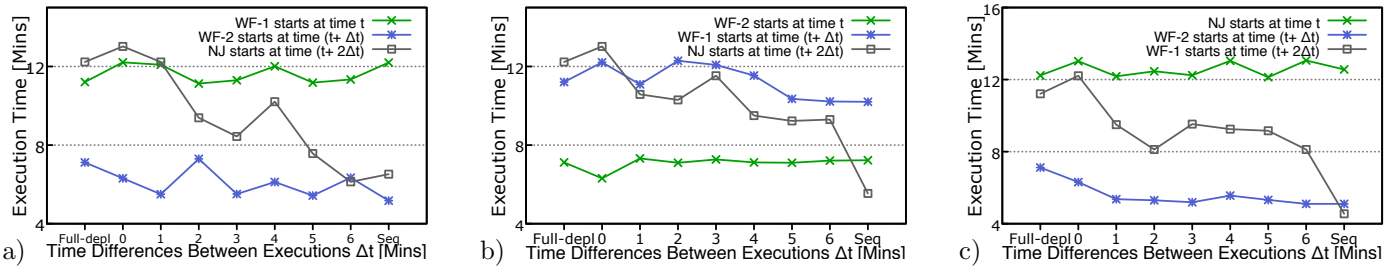

Figure 14: Delayed execution time for different workflows ran on different Clouds ordered in three ways: a) $WF1 \rightarrow WF2 \rightarrow NJ$, b) $WF2 \rightarrow WF1 \rightarrow NJ$, c) $NJ \rightarrow WF2 \rightarrow WF1$.

ecution time.

5. Related Work

We divide research related to our work into two areas. One focuses on the reproducibility and sharing of scientific workflows. The other involves the optimisation of workflow deployment.

5.1. Scientific Workflow Reproducibility and Sharing

Workflow reproducibility and repeatability have been discussed in a number of studies, such as [23] and [4], and are considered to be an essential part of the computational scientific method.

Various attempts have been proposed in the literature to address repeatability and reproducibility of scientific workflows. Most of them follow one of two directions. They either use physical preservation/conservation in which a workflow and its components are packaged and shared together, or logical preservation/conservation that describes a workflow and all its components in a form of a recipe which others can follow to re-create the experiment.

To implement the packaging of workflows different tools and approaches have been proposed and developed such as ReproZip [10] and virtualization mechanisms [24, 25, 26, 27]. There are also efforts to reproduce the workflow using container-based virtualization [28] Despite packaging mechanisms allowing workflows to be re-executed, i.e. allow repeatability, they usually produce large packages that are difficult and costly to distribute publicly. Further, they often do not convey a detailed and structured description of the entire computation, relevant dependencies and execution environments, which would help in understanding the package contents.

The logical preservation techniques focus on capturing all the details required to repeat and potentially reproduce a scientific workflow. They include various approaches to describe workflows such as semantic-based technique [29], capturing the provenance information of the workflow results [30, 31] and using Research Objects as a descriptor [7]. However, other studies have shown that sharing only the specification of a workflow is not enough to ensure its successful reproducibility [7] [29]. The specification-based mechanisms provide various details that can help in understanding the workflow and its components. Yet, they are still insufficient when some of the required dependencies change or become unavailable. For this reason the integration of workflow specification and component description alongside a portable packaging mechanism that facilitates sharing is of fundamental importance.

A number of existing workflow management systems (WfMS) provide a way to share their workflows together with related components. Galaxy [9] enables sharing and publishing of workflows and related objects, such as datasets and tools, via web. Pegasus workflows are described and can be shared using their internal abstract format called the DAX [32]. Taverna [33] and Kepler [34] workflows can be shared via myExperiment [35], a public repository to exchange workflows, computational descriptions and visu-

alizations of their components. However, all these methods are limited as they only allow the structure and description of the workflow to be shared, and still require manual preparation of the execution environment. In contrast, our framework provides the means to automatically provision the workflow execution environment as part of the workflow enactment process. Thus, our shared workflows not only convey the structure and description details, but primarily, are ready-to-run entities which can be deployed and enacted automatically on a set of generic computing nodes, e.g. in the Cloud.

Recently, the RAMP framework [36] has been developed to help the Machine Learning community promote the reproducibility and fairness of the building and evaluation of predictive models. Although RAMP is dedicated specifically to Machine Learning, some concepts resemble our work. Namely, bundle, starting kit and the test submission script are conceptually similar to our TOSCA-based workflow descriptor, test workflow and one-click deployment script. In both cases the intention is to deliver transparent and reproducible software, and lower the entry barrier to experimenters. And despite the RAMP framework has been developed independently and for specific purpose, it clearly reinforces the need to combine a variety of tools and techniques to improve the reproducibility, automate experiment set-up and ease experimentation by providing test examples.

5.2. Workfow Deployment Optimization

We have shown how facilitating the sharing and re-use of deployable workflow components can support optimization of the workflow deployment process. Existing efforts carried out in the field of deployment optimization are often related to making decisions concerning the best models for applications deployment and deployment planning, i.e. finding the optimal placement of components over computational nodes prior to the execution [37, 38, 39, 40]. In contrast, the approach discussed in this paper focuses on the automatic optimization of the component provisioning at deployment time.

There are a few solutions that address the optimization of the provisioning of workflows and other distributed applications. Some of the WfMSes support optimization for the workflow provisioning process by caching various workflow components. e-Science Central [21] is a Cloud-oriented WfMS in which a workflow comprises a set of interconnected blocks. During workflow execution, any blocks and libraries required by the workflow are downloaded from the server on demand and cached in the execution environment [41]. Then, they all are available for later re-use to execute the same or a new workflow. We also implement the caching of essential workflow components such as tasks artifacts and dependency packages in the execution environment. However, our framework is more flexible as it does not impose rigid constraints on the structure of the block and library. We use a well-established containerization technique to package workflow components, and facilitate building the components from TOSCA descriptors. Additionally, we offer the three levels of cache to speed-up workflow enactment across different users and communities.

An interesting approach to provisioning optimization is presented by Vukojevic-Haupt et al. [42]. The authors introduced an on-demand provisioning of services to reduce the cost and time of existing service provisioning and deprovisioning techniques. Once a service has been started, it is kept running for subsequent use. Our approach is different and more suitable for the enactment of workflows. Our tasks are packaged once and then the image is reused by others in their workflows. There is no shared state between workflows yet subsequent provisioning actions are realised very effectively by the containerization platform.

Recently, the Common Workflow Language (CWL) [43] has been proposed to describe analysis workflows and tools. The main intent is to make them portable and scalable across a variety of environments. CWL can be used to

wrap existing command line tools and enable developers and scientists to share them. Tools and workflows described using CWL can optionally use Docker containers with the dependencies required by the tools already pre-installed. However, despite its intention to provide a generic description for workflow tasks, CWL does not offer the ability to include the complete stack of the execution environment. Furthermore, the dependencies required to execute tasks have to be delivered manually not automatically and on-demand. In this respect CWL relies on Docker as a mechanism to capture the installation and execution of tasks and dependencies.

Our approach also uses Docker but automates the provisioning of tasks and their dependencies, which is done on-demand during the deployment time. Additionally, our framework can automatically create workflow and task images, and enables their sharing as TOSCA-based descriptors and/or ready-to-run components. It is therefore more coherent because workflows and tasks are described in the same way, while Docker is only a means of packaging them for later re-use.

5.3. Detailed Comparison with Selected Solutions

To better illustrate how our approach to sharing and performance optimisation of reproducible workflows compares with other solutions, we summarized in Tab. 5 the key differentiating aspects. For each of the systems we consider how they address the aspects in relation to workflow and task, separately.

Firstly, we note that our approach (similarly to systems based on the CWL specification) uses a common language to define both workflow and tasks. It makes the overall workflow definition more consistent and easier to comprehend by users. All other non-CWL WfMSes use their own proprietary languages, often different for workflows and tasks, which creates an additional barrier to learning and sharing. The exception is Taverna which combines workflows from tasks implemented as web services, and so the actual way how tasks are defined depends on the technology used. Thus, despite Taverna makes sharing of tasks/services very easy, the main barrier for the user is the need to implement, deploy and maintain them as a web service.

Secondly, looking at the packaging mechanism we note that our solution is unique in that it can turn a workflow into a compact executable package (Docker image). This is possible when the user decides to use the single-container configuration; the benefits of this approach are discussed earlier in Sec. 2.5. An attempt to package Galaxy workflows into executable units is described in [44]. It imposes, however, high overheads as the workflow is packed together with the whole Galaxy instance into a VM image. Other efforts to turn workflows into executable packages include the use of Research Objects [45] and DataONE packaging [46].

When using the multi-container configuration, we rely directly on the TOSCA description to run and share our workflows, and so no packaging mechanism is available. It is similar to how other systems operate.

Task packaging is in the most cases implemented using Docker which proved to be a very efficient and lightweight solution. Galaxy stands out in this respect by offering users a three layer packaging approach which involves: the Conda package manager, a container technology like Docker, Singularity and rkt, and VM images [47]. Users can decide how to package their software depending on the desired level of isolation.

However, only our approach offers a fully automated mechanism to task packaging. The presented versioning and naming scheme combined with the AIC relieves users from image management and facilitates sharing. To some extent Arvados, `cwltool` and Galaxy also allow images to be generated automatically if the task definition is augmented with Dockerfile. But images generated in this way are used only for the purpose of particular workflow execution rather than for sharing with other users.

Table 5: Comparison between selected approaches to sharing and optimisation of workflows.

		description language	packaging mechanism		artifact cache	automated updates
Arvados	workflow	CWL	–	–	–	–
	task	CWL/Dockerfile	Docker	automated	2-level	–
cwltool	workflow	CWL	–	–	–	–
	task	CWL/Dockerfile	Docker	automated	2-level	–
e-Science Central	workflow	internal XML	–	–	–	yes
	task	proprietary XML	Zip Archive	manual	2-level	–
Galaxy	workflow	proprietary JSON	–/third-party	–	–	–
	task	proprietary XML	Conda/Docker, Singularity, rkt/VM	automated	1-level	–
Pegasus	workflow	DAX	–	–	–	yes
	task	DAX/TC	–/Docker	manual	–	–
Taverna	workflow	SCUFL2	–	–	–	yes
	task	web-service API	web service	manual	–	–
The proposed approach	workflow	TOSCA	–/Docker	–/automated	–/3-level	yes
	task	TOSCA	Docker	automated	3-level	yes

Thirdly, we compared systems in terms of the caching technique they use to optimise workflow enactment and sharing. We proposed to use the 3-level cache such that the on-host cache maintained by Docker is supported by the second-, organisation-level cache and the top-level repository. All they play an important role in the optimisation and/or sharing, as shown in the Evaluation section, yet other systems offer at most two levels of caching. Interestingly, Arvados allows also containers to be shared between tasks in case they use the same configuration parameters. This form of caching may further improve enactment although Docker containers are already lightweight and their start imposes only small overhead in range of hundreds of milliseconds.

Also Pegasus is different with regard to task caching as it relies on the Transformation Catalog to bind tasks with their executable software. Then, depending on the task and execution environment the software have to be installed manually, may be staged in from a remote location or pulled from the Docker repository. Taverna, instead, relies on external web services, and so does not need to cache any software. On the other hand, relying on users to deliver web services may negatively impact the overall workflow performance.

The final aspect we consider is the ability to automatically update tasks while maintaining reproducibility of workflows. Here, we distinguish two cases: whether a task update is automatically passed to the workflow level, and whether a task update makes the task package to be automatically refreshed. Our solution can deliver both. Some of the other systems can maintain only workflows up to date. In Taverna this is hidden from the workflow by the web service API. Pegasus workflows can specify tasks with a version pattern, and so allow updated tasks to be used automatically. e-Science Central enables users to indicate a specific version of the task to be used or its 'latest' version. However, all of them rely on the user to repackage the task software following the update.

6. Conclusion and Future Work

In this paper we have presented the design and evaluation of the framework that supports portable modeling, on-demand automatic provisioning, re-usability and reproducibility of scientific workflows. The key feature of our framework is its ability to combine logical and physical reproducibility techniques, which significantly improves the overall reproducibility of our workflows. To lower the costs related to keeping workflows reproducible we have concentrated in this paper on the methods to improve runtime performance. Specifically, we introduced optimizations of the workflow provisioning and enactment process that lead to the substantial reduction of the workflows runtime. We were able, for example, to reduce the runtime of the *NJ* workflow by about 20 times, from over 30 minutes to only 96 seconds.

The main features of the proposed optimizations include a new algorithm for the automated management of workflow and task images. We combined it with a 3-level cache of workflow components, task artifact and dependencies, which helped reduce the time required to upload and download task images by a factor of 2–3. The use of the cache improves not only the runtime performance but also helps in sharing tasks and workflows, and can reduce the initial deployment and image creation times. The basis for the optimization is the fact that workflows usually have many common tasks and dependencies, so adding a cache can effectively accelerate provisioning and enactment.

Finally, the proposed techniques incorporate the tracking of changes in workflow tasks to effectively select the most suitable image for provisioning. The framework's integration with a version control system allows backward-compatible changes to be distributed transparently and on-demand across workflow engines. Thus, our system helps developers to streamline the process of building and distributing their workflows and tasks. At the same time, it also benefits end-users as their workflows automatically receive compatible updates; all without breaking the reproducibility of workflows. Moreover, tracking changes is also useful for the concurrent provisioning of workflows and tasks. We demonstrated these effects through an extensive set of experiments.

In the future we would like to investigate to what extent our framework can be used to model, deploy and reproduce existing workflows designed in other scientific workflow management systems or languages like Pegasus, Taverna, and CWL. In addition, as our framework can be used to deploy the whole workflow on different Cloud and local environments, a natural extension is to support the enactment of large-scale, distributed workflows that span multiple Clouds - for example public and private Clouds.

7. Acknowledgement

This work was partially supported by the ReComp project, EPSRC grant EP/N01426X/1, and the National Natural Science Foundation of China (No. 61701444).

References

[1] A. Barker, J. Van Hemert, Scientific workflow: a survey and research directions, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (Eds.), Parallel Processing and Applied Mathematics, Vol. 4967, Springer, 2007, pp. 746–753. doi:10.1007/978-3-540-68111-3_78.

[2] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-Science: An overview of workflow system features and capabilities, Future Generation Computer Systems 25 (5) (2009) 528–540. doi:10.1016/j.future.2008.06.012.

[3] G. Juve, E. Deelman, Scientific Workflows in the Cloud, in: M. Cafaro, G. Aloisio (Eds.), Grids, Clouds and Virtualization, Springer London, London, 2011, pp. 71–91. doi:10.1007/978-0-85729-049-6_4.

[4] S. Arabas, M. R. Bareford, L. R. de Silva, I. P. Gent, B. M. Gorman, M. Hajiarabderkani, T. Henderson, L. Hutton, A. Konovalov, L. Kotthoff, et al., Case Studies and Challenges in Reproducibility in the Computational Sciences, [online], arXiv:1408.2123 (2014).

[5] K. M. Hettne, K. Wolstencroft, K. Belhajjame, C. A. Goble, E. Mina, H. Dharuri, D. De Roure, L. Verdes-Montenegro, J. Garrido, M. Roos, Best Practices for Workflow Design: How

to Prevent Workflow Decay, in: A. Paschke, A. Burger, P. Romano, M. S. Marshall, A. Splendiani (Eds.), Semantic Web Applications and Tools for Life Sciences 2012, Vol. 952, 2012.

[6] B. K. Beaulieu-Jones, C. S. Greene, Reproducibility of computational workflows is automated using continuous analysis, Nature Biotechnology 35 (2017) 342–346. doi:10.1038/nbt.3780.

[7] K. Belhajjame, J. Zhao, D. Garijo, M. Gamble, K. Hettne, R. Palma, E. Mina, O. Corcho, J. M. Gómez-Pérez, S. Bechhofer, et al., Using a suite of ontologies for preserving workflow-centric research objects, Web Semantics: Science, Services and Agents on the World Wide Web 32 (2015) 16–42.

[8] S. Cohen-Boulakia, K. Belhajjame, O. Collin, J. Chopard, C. Froidevaux, A. Gaignard, K. Hinsen, P. Larmande, Y. Le Bras, F. Lemoine, F. Mareuil, H. Ménager, C. Pradal, C. Blanchet, Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities, Future Generation Computer Systems 75 (2017) 284–298. doi:10.1016/j.future.2017.01.012.

[9] E. Afgan, D. Baker, B. Batut, M. van den Beek, D. Bouvier, M. Čech, J. Chilton, D. Clements, N. Coraor, B. A. Grüning, A. Guerler, J. Hillman-Jackson, S. Hiltemann, V. Jalili, H. Rasche, N. Soranzo, J. Goecks, J. Taylor, A. Nekrutenko, D. Blankenberg, The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update, Nucleic Acids Research 46 (W1) (2018) W537–W544. doi:10.1093/nar/gky379.

[10] F. Chirigati, D. Shasha, J. Freire, ReproZip: Using Provenance to Support Computational Reproducibility, in: 5th USENIX Workshop on the Theory and Practice of Provenance, USENIX, Lombard, IL, 2013.

[11] E. Deelman, K. Vahi, M. Rynge, G. Juve, R. Mayani, R. F. da Silva, Pegasus in the Cloud: Science Automation through Workflow Technologies, IEEE Internet Computing 20 (1) (2016) 70–76. doi:10.1109/MIC.2016.15.

[12] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, C. Goble, The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud, Nucleic Acids Research 41 (W1) (2013) W557–W561. doi:10.1093/nar/gkt328.

[13] L. Wang, Z. Lu, P. Van Buren, D. Ware, J. Hancock, Sciapps: A cloud-based platform for reproducible bioinformatics workflows, Bioinformatics 1 (2018) 4.

[14] R. Qasha, J. Cała, P. Watson, A framework for scientific workflow reproducibility in the cloud, in: 2016 IEEE 12th International Conference on e-Science (e-Science), 2016, pp. 81–90. doi:10.1109/eScience.2016.7870888.

[15] TOSCA Simple Profile in YAML Version 1.2, [online] (2018). URL https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

[16] R. Qasha, J. Cała, P. Watson, Towards Automated Workflow Deployment in the Cloud Using TOSCA, in: 2015 IEEE 8th International Conference on Cloud Computing, 2015, pp. 1037–1040. doi:10.1109/CLOUD.2015.146.

[17] J. F. Pimentel, J. Freire, V. Braganholo, L. Murta, Tracking and analyzing the evolution of provenance from scripts, in: M. Mattoso, B. Glavic (Eds.), Provenance and Annotation of Data and Processes, Springer International Publishing, Cham, 2016, pp. 16–28. doi:10.1007/978-3-319-40593-3_2.

[18] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, TOSCA: Portable Automated Deployment and Management of Cloud Applications, in: A. Bouguettaya, Q. Z. Sheng, F. Daniel (Eds.), Advanced Web Services, Springer New York, New York, NY, 2014, pp. 527–549. doi:10.1007/978-1-4614-7535-4_22.

[19] T. Binz, G. Breiter, F. Leyman, T. Spatzier, Portable Cloud Services Using TOSCA, IEEE Internet Computing 16 (3) (2012) 80–85. doi:10.1109/MIC.2012.43.

[20] J. Cała, E. Marei, Y. Xu, K. Takeda, P. Missier, Scalable and efficient whole-exome data processing using workflows on the cloud, Future Generation Computer Systems 65 (2016) 153–168, special Issue on Big Data in the Cloud. doi:https://doi.org/10.1016/j.future.2016.01.001.

[21] H. Hiden, S. Woodman, P. Watson, J. Cała, Developing cloud applications using the e-Science Central platform, Phil. Trans. R. Soc. A 371 (1983). doi:10.1098/rsta.2012.0085.

[22] J. Mertz, I. Nunes, Understanding application-level caching in web applications: A comprehensive introduction and survey of state-of-the-art approaches, ACM Comput. Surv. 50 (6) (2017) 98:1–98:34. doi:10.1145/3145813.

[23] J. Zhao, J. M. Gómez-Pérez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, C. Goble, Why workflows break – Understanding and combating decay in Taverna workflows, in: E-Science (e-Science), 2012 IEEE 8th International Conference on, IEEE, 2012, pp. 1–9. doi:10.1109/eScience.2012.6404482.

[24] M. Dahlö, F. Haziza, A. Kallio, E. Korpelainen, E. Bongcam-Rudloff, O. Spjuth, BioImg.org: A Catalog of Virtual Machine Images for the Life Sciences, Bioinformatics and Biology Insights 9 (2015) 125–128. doi:10.4137/BBI.S28636.

[25] V. Stodden, F. Leisch, R. D. Peng, Implementing Reproducible Research, 1st Edition, CRC Press, 2014.

[26] B. Howe, Virtual Appliances, Cloud Computing, and Reproducible Research, Computing in Science & Engineering 14 (4) (2012) 36–41. doi:10.1109/MCSE.2012.62.

[27] F. Jiang, C. Castillo, C. Schmitt, A. Mandal, P. Ruth, I. Baldin,

Enabling Workflow Repeatability with Virtualization Support, in: Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science, WORKS '15, ACM, New York, NY, USA, 2015, pp. 8:1–8:10. doi:10.1145/2822332.2822340.

[28] A. M. Kintsakis, F. E. Psomopoulos, A. L. Symeonidis, P. A. Mitkas, Hermes: Seamless delivery of containerized bioinformatics workflows in hybrid cloud (HTC) environments, SoftwareX 6 (2017) 217–224. doi:10.1016/j.softx.2017.07.007.

[29] I. Santana-Perez, R. F. da Silva, M. Rynge, E. Deelman, M. S. Pérez-Hernández, O. Corcho, Reproducibility of execution environments in computational science using semantics and clouds, Future Generation Computer Systems 67 (2017) 354–367. doi:10.1016/j.future.2015.12.017.

[30] P. Missier, S. Woodman, H. Hiden, P. Watson, Provenance and data differencing for workflow reproducibility analysis, Concurr. Comput.: Pract. Exper. 28 (4) (2016) 995–1015. doi:10.1002/cpe.3035.

[31] Q. Pham, T. Malik, I. Foster, Auditing and Maintaining Provenance in Software Packages, in: B. Ludäscher, B. Plale (Eds.), Provenance and Annotation of Data and Processes, Vol. 8628 of Lecture Notes in Computer Science, Springer International Publishing, Cham, 2015, pp. 97–109. doi:10.1007/978-3-319-16462-5_8.

[32] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (2015) 17–35. doi:10.1016/j.future.2014.10.008.

[33] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, C. Goble, Taverna, reloaded, in: M. Gertz, B. Ludäscher (Eds.), Scientific and Statistical Database Management, Vol. 6187 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 471–481. doi:10.1007/978-3-642-13818-8_33.

[34] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system, Concurr. Comput.: Pract. Exper. 18 (10) (2006) 1039–1065. doi:10.1002/cpe.994.

[35] C. A. Goble, J. Bhagat, S. Aleksejevs, D. Cruickshank, D. Michaelides, D. Newman, M. Borkum, S. Bechhofer, M. Roos, P. Li, et al., myExperiment: a repository and social network for the sharing of bioinformatics workflows, Nucleic Acids Research 38 (suppl_2) (2010) W677–W682. doi:10.1093/nar/gkq429.

[36] B. Kégl, A. Boucaud, M. Cherti, A. Kazakci, A. Gramfort, G. Lemaitre, J. Van den Bossche, D. Benbouzid, C. Marini, The RAMP framework: from reproducibility to transparency in the design and optimization of scientific workflows, [online], accessed Nov/2018 (2018).
URL https://openreview.net/pdf?id=Syg4NHz4eQ

[37] Z. Wen, J. Cała, P. Watson, A. Romanovsky, Cost effective, reliable and secure workflow deployment over federated clouds, IEEE Transactions on Services Computing 10 (6) (2017) 929–941. doi:10.1109/TSC.2016.2543719.

[38] S. Holl, O. Zimmermann, M. Palmblad, Y. Mohammed, M. Hofmann-Apitius, A new optimization phase for scientific workflow management systems, Future Generation Computer Systems 36 (2014) 352–362. doi:10.1016/j.future.2013.09.005.

[39] K. Maheshwari, E.-S. Jung, J. Meng, V. Morozov, V. Vishwanath, R. Kettimuthu, Workflow performance improvement using model-based scheduling over multiple clusters and clouds, Future Generation Computer Systems 54 (2016) 206–218. doi:10.1016/j.future.2015.03.017.

[40] G. Kougka, A. Gounaris, A. Simitsis, The many faces of data-centric workflow optimization: a survey, International Journal of Data Science and Analytics 6 (2) (2018) 81–107. doi:10.1007/s41060-018-0107-0.

[41] J. Cała, H. Hiden, S. Woodman, P. Watson, Cloud computing for fast prediction of chemical activity, Future Generation Computer Systems 29 (7) (2013) 1860–1869. doi:10.1016/j.future.2013.01.011.

[42] K. Vukojevic-Haupt, S. G. Sáez, F. Haupt, D. Karastoyanova, F. Leymann, A middleware-centric optimization approach for the automated provisioning of services in the cloud, in: Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on, IEEE, 2015, pp. 174–179. doi:10.1109/CloudCom.2015.86.

[43] P. Amstutz, M. R. Crusoe, N. Tijanić, Common Workflow Language Specifications, v1.0.2, [online], accessed Nov/2018.
URL https://www.commonwl.org/v1.0/

[44] F. Bartusch, M. Hanussek, J. Krüger, Containerization of Galaxy Workflows increases Reproducibility, in: Proc. of 4th bwHPC Symposium, Tübingen, 2017, pp. 16–19.

[45] S. Soiland-Reyes, CWL Research Objects, accessed 22/Mar/2019 (2018).
URL http://slides.com/soilandreyes/2018-01-26-cwl-ro-elixir

[46] B. Mecum, M. B. Jones, D. Vieglais, C. Willis, Preserving Reproducibility: Provenance and Executable Containers in DataONE Data Packages, in: 2018 IEEE 14th International Conference on e-Science (e-Science), IEEE, 2018, pp. 45–49. doi:10.1109/eScience.2018.00019.

[47] B. Grüning, J. Chilton, J. Köster, R. Dale, N. Soranzo, M. van den Beek, J. Goecks, R. Backofen, A. Nekrutenko, J. Taylor, Practical Computational Reproducibility in the Life Sciences, Cell Systems 6 (6) (2018) 631–

635. doi:10.1016/j.cels.2018.03.014.

Appendix A. The structure of tested workflows

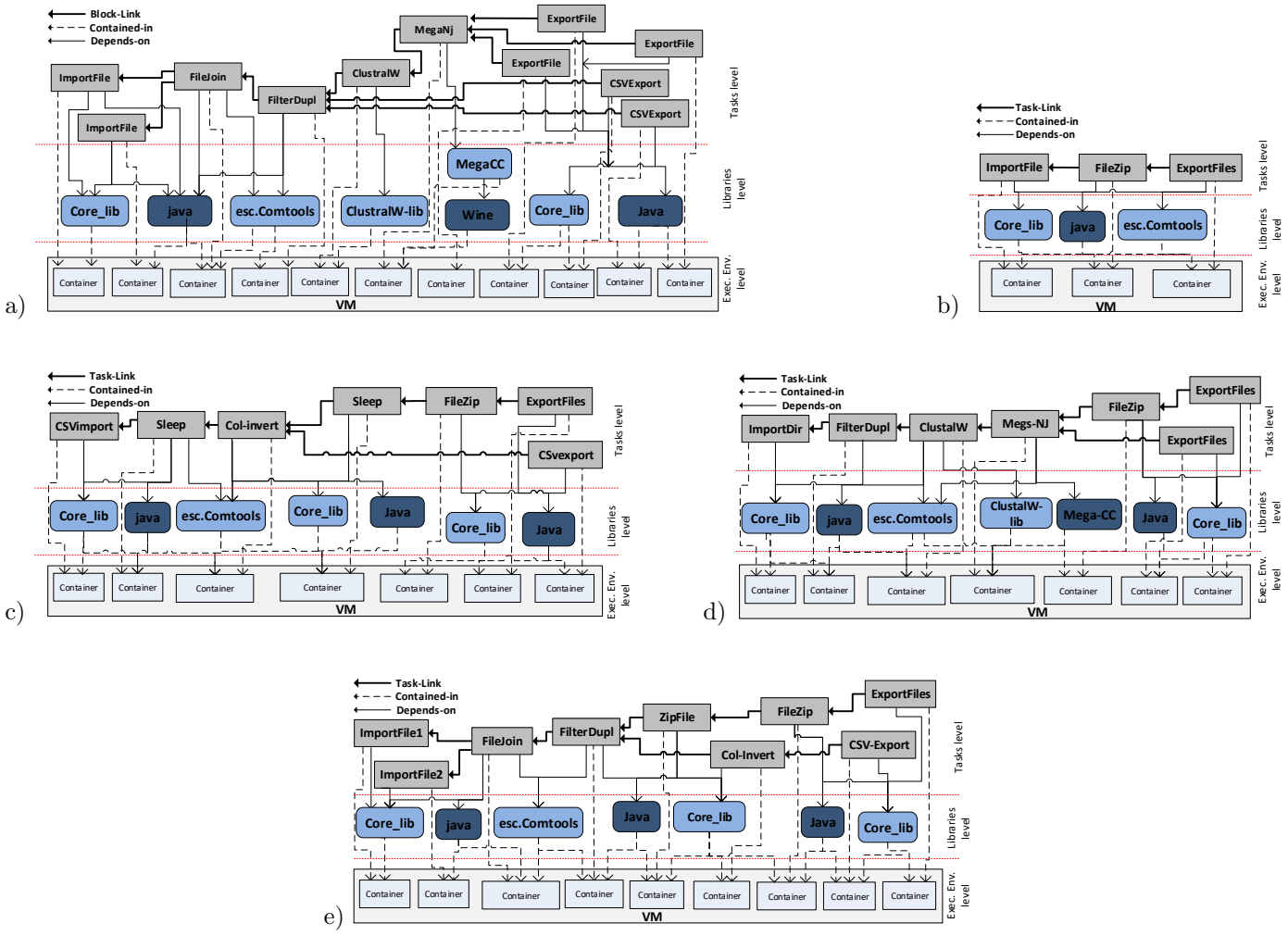

Figure A.15: The structure of the tested workflows in the multi-container configuration: a) Neighbour Joining (NJ), b) FileZip (FZ), c) Column Inverter (CI), d) WF-1 and e) WF-2.